HEWLETT-PACKARD COMPANY
Intellectual Property Administration
P.O. Box 272400
Fort Collins, Colorado 80527-2400

PATENT APPLICATION

ATTORNEY DOCKET NO. 10011596-1

IN THE
UNITED STATES PATENT AND TRADEMARK OFFICE

Inventor(s): Christophe de Dinechin et al.

Confirmation No.: 5117

Application No.: 09/873,875

Examiner: Lillian Vo

Filing Date: 6/4/2001

Group Art Unit: 2195

Title: CONTEXT-CORRUPTING CONTEXT SWITCHING

Mail Stop Appeal Brief-Patents
Commissioner For Patents
PO Box 1450
Alexandria, VA 22313-1450

TRANSMITTAL OF APPEAL BRIEF

Transmitted herewith is the Appeal Brief in this application with respect to the Notice of Appeal filed on 3/14/2008.

☐ The fee for filing this Appeal Brief is $510.00 (37 CFR 41.20).

☒ No Additional Fee Required.

(complete (a) or (b) as applicable)

The proceedings herein are for a patent application and the provisions of 37 CFR 1.136(a) apply.

☐ (a) Applicant petitions for an extension of time under 37 CFR 1.136 (fees: 37 CFR 1.17(a)-(d)) for the total number of months checked below:

☐ 1st Month $120	☐ 2nd Month $460	☐ 3rd Month $1050	☐ 4th Month $1640

☐ The extension fee has already been filed in this application.

☒ (b) Applicant believes that no extension of time is required. However, this conditional petition is being made to provide for the possibility that applicant has inadvertently overlooked the need for a petition and fee for extension of time.

Please charge to Deposit Account 08-2025 the sum of $ 00 . At any time during the pendency of this application, please charge any fees required or credit any over payment to Deposit Account 08-2025 pursuant to 37 CFR 1.25. Additionally please charge any fees to Deposit Account 08-2025 under 37 CFR 1.16 through 1.21 inclusive, and any other sections in Title 37 of the Code of Federal Regulations that may regulate fees.

Respectfully submitted,

Christophe de Dinechin et al.

By /Hugh Gortler #33890/

Hugh P. Gortler

Attorney/Agent for Applicant(s)

Reg No. : 33,890

Date : 3/19/2008

Telephone : (949) 454-0898

Docket No. 10011596-1

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE
BEFORE THE BOARD OF PATENT APPEALS
AND INTERFERENCES

APPEAL NO. ________________

In re Application of:
Christophe de Dinechin et al.

Confirmation No. 5117

Serial No. 09/873,875
Filed: June 4, 2001

Examiner: Lillian Vo
Art Unit: 2195

For: **CONTEXT-CORRUPTING CONTEXT SWITCHING**

APPEAL BRIEF

Hugh P. Gortler, Esq.

Hewlett-Packard Company
Intellectual Property Administration
P.O. Box 272400
Fort Collins, Colorado 80527-2400

(949) 454-0898

INDEX

1. REAL PARTY IN INTEREST

The real party in interest is the assignee, Hewlett-Packard Development Company.

2. RELATED APPEALS AND INTERFERENCES

No appeals or interferences are known to have a bearing on the Board's decision in the pending appeal.

3. STATUS OF CLAIMS

Claims 1-26 are pending.

Claims 1-26 are rejected.

The rejections of claims 1-5, 9, 11-16, 20 and 22-26 are being appealed.

4. STATUS OF AMENDMENTS

No amendments have been filed since the final office action dated April 6, 2005.

5. SUMMARY OF CLAIMED SUBJECT MATTER

An operating system (OS) can be run as application on a computer using a virtual machine. The OS that is run as the application is referred to as the "guest OS," and the underlying OS is referred to as the "host OS." For example, a Windows-based OS can be run as an application on top of a host Linux OS. Running the Windows-based OS as an application allows programs written for a Windows environment to be run on top of the host Linux OS.

The guest OS usually has a different "context" than the host OS. A context embodies the accessible state of the computer's central processing unit (CPU). The context includes values of CPU registers.

The virtual machine can allow the guest OS and the host OS to run concurrently by properly restoring the context of the guest OS whenever control is transferred from the host OS to the guest OS, and by similarly restoring the context of the host OS when control is transferred from the guest OS to the host OS. This process is called "context switching."

Certain CPU architectures, such as IA-64, have prohibitions against, and difficulties with, completely saving and restoring the entire context of a guest OS or a host OS. Specifically, certain registers containing context might be corrupted during context switching. If the entire context cannot be saved, the guest OS or host OS might behave incorrectly and crash.

A simple example in paragraph 17 of the application illustrates a problem that such architectures have with context switching. Context of a host OS is contained in registers X, I, J and K. During context switching, register X is used to store an address that indicates where the context will be saved. However, the register X is overwritten during context switching, whereby the context of register X is lost before it can be saved.

This problem is overcome by the method of claim 1, the method of claim 11, the apparatus of claim 12, and the software of claim 22.

Base claim 1

Claim 1 recites a method of switching context on a processor (element 110 in Figure 1). Referring to Figure 2 and paragraphs 20-21 of the specification, the method comprises saving the context under software control using an inconsequential register (312); and preventing the processor from changing the context while the context is being saved (310).

The inconsequential register can be used to store context, for example, by storing an address that indicates where the context will be saved. This example is described in paragraph 23 of the application.

According to paragraph 21 of the specification, an inconsequential register is a register that is not used by the host OS at a predetermined interruption point (PIP). A point can be predetermined if the context is saved under software control (which is synchronous) instead of hardware control (which is asynchronous). According to paragraph 22 of the application, since the context of a host OS is saved at the PIP, it is known which registers the host OS uses and which registers the host OS does not use. Therefore, the inconsequential register(s) at the PIP can be identified. The inconsequential registers can be corrupted by a virtual machine application without affecting the host OS.

Dependent claim 3

Claim 3, which depends from claim 1, recites that the context is saved at a predetermined interruption point. A predetermined interruption point is described in paragraph 21 of the specification. This point can be determined if the context switching is performed under software (synchronous) control.

Dependent claim 25

Claim 25, which depends from claim 1, recites that the inconsequential register does not store context at a predetermined interruption point. Paragraph 21 of the specification provides support for this feature.

Dependent claim 26

Claim 25, which depends from claim 1, recites that the context is stored in memory other than the inconsequential register. Paragraph 23 provides an example that supports this feature (the inconsequential register stores an address of the memory location where the context will be saved).

Base claim 22

Reference is made to Figures 1-2 and paragraphs 20-21 of the specification. Claim 22 recites software (212) comprising instructions for commanding a processor to switch context by saving the context under software control using an inconsequential register of the processor as temporary storage (312); and preventing the processor from changing the context while the context is being saved (310).

Base claim 11

Reference is made to Figures 1-2 and paragraphs 20-22 of the specification. Claim 11 recites a method of switching context between a host OS (210) and a virtual machine (212) on a processor (110). The processor (110) has privileged registers and access to other memory (112). Referring additionally to Figure 3 and paragraph 29, the method comprises giving the virtual machine access to the privileged registers (point A in Figure 3); using at least one privileged register as temporary storage to save the context in the other memory at a

predetermined interruption point (between points A and B in Figure 3); and preventing the processor from changing the context while the context is being saved (310). The virtual machine controls the context switch.

<u>Base claim 12</u>

Reference is made to Figures 1-2 and paragraphs 20-21 of the specification. Claim 12 recites apparatus comprising a processor (110) including a plurality of registers (110) and a virtual machine application (212). The virtual machine application (212) commands the processor (110) to switch context by saving the context under software control using an inconsequential register of the processor as temporary storage (312). The virtual machine application also prevents the processor from changing the context while the context is being saved (310).

6. <u>GROUNDS OF REJECTION TO BE REVIEWED ON APPEAL</u>

a. Claim 25 is rejected under 35 U.S.C. §112, second paragraph, as being indefinite.

b. Claim 26 is objected to as being indefinite.

c. Claims 1-3, 9 and 22-26 are rejected under 35 U.S.C §103 as being unpatentable over Chatterjee U.S. Patent No. 5,872,963.

d. Claims 4-5, 11-16 and 20 are rejected under 35 U.S.C §103 over Chatterjee in view of Bugnion U.S. Patent No. 6,496,847.

e. Claim 22 is rejected under 35 U.S.C §101.

7. <u>ARGUMENTS</u>

I

REJECTION OF CLAIM 25 UNDER 35 USC 112, SECOND PARAGRAPH AS BEING INDEFINITE

Claim 1 recites “saving the context under software control using an inconsequential register.” Claim 25 recites that “the inconsequential register does not store context at a predetermined interruption point.”

The office action alleges that claim 25 is indefinite because of a supposed contradiction in claim language with its parent claim. The office action interprets “saving context ... using an inconsequential register” **<u>to only mean</u>** storing context in that inconsequential register. The interpretation is not supported by the specification.

Consider the example in paragraph 23 of the specification. In that example, an inconsequential register I is used to store the address at which the context will be stored, and context in registers J, K and X is saved at that address. Thus, the inconsequential register I does not store context. Rather, by storing the address, the inconsequential register ***is used*** to store context.

The office action cites MPEP 2111.01, Claim Interpretation. However, MPEP 2111 states the PTO “determines the scope of claims in patent applications not solely on the basis of the claim language, but upon giving claims their broadest reasonable construction ‘in light of the specification as it would be interpreted by one of ordinary skill in the art.’ MPEP 2111.01 states “the words of a claim must be given their ‘plain meaning’ unless such meaning is inconsistent with the specification. The interpretation that “saving context ... using an inconsequential register” <u>can only mean</u> storing context in that inconsequential register is

inconsistent with the specification. Therefore, the ‘112 rejection is based on an unreasonable interpretation. Accordingly, the ‘112 rejection of claim 25 should be withdrawn.

II
OBJECTION TO CLAIM 26 AS BEING INDEFINITE

The reason for this objection is identical to the reason for the '112 rejection of claim 25: an alleged contradiction of a dependent claim with its parent claim. In addition, this objection was previously raised as a '112 rejection. Therefore, it is addressed in this appeal brief.

Claim 1 recites "saving the context under software control using an inconsequential register." Claim 26 recites that context is stored in memory other than the inconsequential register.

In making this objection, the office action interprets "saving context … using an inconsequential register" **to only mean** storing context in that inconsequential register. The interpretation is not supported by the specification.

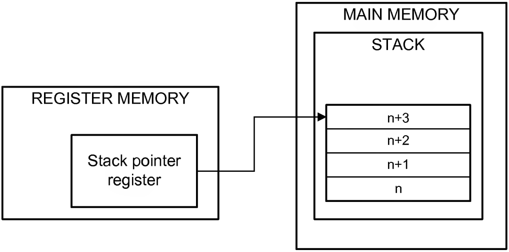
Consider the example in paragraph 23 of the specification. In that example, an inconsequential register I is used to store the address at which the context will be stored, and context in registers J, K and X is saved at that address. Thus, the inconsequential register I does not store context. Rather, by storing the address, the inconsequential register ***is used*** to store context.

MPEP 2111.01 states "the words of a claim must be given their 'plain meaning' unless such meaning is inconsistent with the specification. The interpretation that "saving context … using an inconsequential register" can only mean storing context in that inconsequential register is inconsistent with the specification. Therefore, the objection of claim 26 is based on an unreasonable interpretation. Accordingly, the objection should be withdrawn.

III
REJECTION OF BASE CLAIMS 1 AND 22 AND DEPENDENT CLAIMS 2-3, 9 AND 23-26 UNDER 35 U.S.C §103(b) AS BEING UNPATENTABLE OVER CHATTERJEE U.S. PATENT NO. 5,634,046

Chatterjee is irrelevant to claims 1-3, 9 and 22-26.

Chatterjee's Background discusses a stack for an Intel x86 processor (col. 1, lines 54 to col. 2, line 21). The stack consists of a set of memory locations in system memory (col. 1, lines 62-63). The stack functions as a last-in, first-out (LIFO) buffer (col. 2, lines 4-7) in that data is added to the top of the stack (that is, pushed onto the stack), and data is read and removed from the top of the stack (that is, popped off the stack). A stack pointer register, which is part of register memory, points to the memory location of the last data pushed onto the stack (col. 1, line 66 to col. 2, line 1). For the purpose of illustration, the figure below shows a stack pointer register pointing to location n+3.



The stack may be used to store a return address when switching execution from one program to another (col. 2, lines 8-21). Say a program is executing a series of instructions. The instruction being executed is stored in an instruction register. If an interrupt occurs, the address of the instruction being executed is read from the instruction register and temporarily pushed onto the stack. The stack pointer register points to the location of that return address on the stack. The starting address of an interrupt handler is then stored in the instruction register, and the interrupt handler responds to the interrupt. Once the interrupt handler completes its operation, the address pointed to by the stack pointer register (the return address) is popped off the stack and written back to the instruction register. The value of the stack pointer register is decremented, and the interrupted program resumes operation.

Chatterjee sees value in using the stack pointer register for general purposes, such as storing operands and the results of operations (col. 2, lines 65-67). Accessing register memory is faster than accessing system memory (col. 1, lines 29-32; and col. 7, lines 55-58).

However, Chatterjee's Background identifies a problem with using the stack register for general purposes in Intel x86 processors (col. 2, lines 22-32). Say an interrupt occurs while the stack pointer register is being used for general purposes (e.g., storing an operand). If the stack pointer register is being used for general purpose at the time of the interrupt, it will not point to the correct location in the stack. Thus, the address in the instruction register will be written to the wrong location on the stack, possibly overwriting other information already on the stack. Chatterjee refers to this problem as "corrupting the stack" (col. 2, lines 29-30).

Chatterjee purportedly prevents stack corruption by exploiting a specific feature of x86 processors: a protected mode of operation (col. 2, lines 42-52). The stack includes different segments, and each segment corresponds to a different privilege level (col. 6, line 52 to col. 7, line 8). Chatterjee sets the privilege level of the interrupt handler to a level that is higher than the executing program's privilege level (col. 7, lines 13-15). According to col. 2, lines 54-60:

> When the interrupt handler is at a higher privilege level than the executing program, the microprocessor switches to the interrupt handler's privilege level on receiving an interrupt. The return address of the executing program is not pushed onto the stack until after switching to the interrupt handler's privilege level. Accordingly, the return address is stored in the stack segment of the interrupt handler's privilege level and not that of the currently executing program.

Thus, Chatterjee appears to achieve his objective (preventing corruption of the stack) by switching from a first stack segment to a second stack segment in order to avoid corrupting the first stack segment.

Chatterjee's operations are performed by an x86 microprocessor in response to asynchronous interrupts. That is, interrupts are handled under hardware control (col. 5, lines 62 to col. 6, line 10).

Chatterjee doesn't address the problem faced by the applicants: the corruption of certain registers containing context during a context switch. Chatterjee is concerned with using a stack pointer register for general purposes.

Chatterjee is concerned with the corruption of a return address in main memory (where the stack is located). Chatterjee is not concerned about the corruption of context in register memory.

Chatterjee has no need to distinguish between registers that are inconsequential and those that are not. Chatterjee's focus is on the stack pointer register, and takes measures to ensure that it doesn't corrupt the stack.

Chatterjee handles interrupts under hardware (asynchronous) control (col. 2, lines 22-23). Chatterjee does not teach or suggest handling interrupts under software (synchronous) control.

Chatterjee is silent about interruption points that can be predetermined. According to Chatterjee, an interrupt can occur at any time (col. 2, lines 22-23).

It follows that Chatterjee does not teach or suggest a method of switching context on a processor, including "saving the context under software control using an inconsequential register."

Chatterjee does not prevent a processor from changing context while the context is being saved. Chatterjee simply operates the microprocessor in protected mode so it does not push a return address of an executing program onto the stack until the stack pointer register has been used for general purposes.

Finally, Chatterjee doesn't describe how his method can be used to prevent context from being corrupted during a context switch. Neither does the office action.

It is not clear why the '103 rejection is raised, since the office action alleges that all features recited in claims 1-3, 9 and 22-26 are disclosed in Chatterjee. That is, the office action does not identify any differences between claim 1 and Chatterjee or indicate why the differences are obvious.

The office action discusses an inconsequential register and interprets it as a type of storage that is not used at a predetermined interrupt point. However, that interpretation is incomplete. According to paragraph 21 of the specification, the inconsequential register does not store context at a predetermined interrupt point.

Chatterjee does not teach, hint or remotely suggest identifying registers at a predetermined interrupt point. Chatterjee suggests it can't be done because interrupts can occur at any time.

And even then, the stack pointer register is always being used, so it wouldn't be considered an inconsequential register. Either the stack pointer register is being used to point to an address in main memory, or it is being used for general purposes.

Now perhaps the office action acknowledges that Chatterjee doesn't disclose a method for performing a context switch. However, it doesn't clearly state it, and it only make a bald conclusion that it is obvious to use the "stack pointer register in place of the claimed inconsequential register.

However, none of the cited documents support that conclusion, and the office action does not support it with any other reasons. The PTO has the burden of establishing a *prima facie* case of obviousness under 35 USC §103. The Patent Office must show some reason to combine the elements with some rational underpinning that would lead an individual of ordinary skill in the art to combine the relevant teachings of the references. *KSR International Co. v. Teleflex Inc.*, No. 04-1350, 127 S. Ct. 1727 (2007); *In re Fine*, 837 F.2d 1071, 1074 (Fed.Cir.1988).

The office action has not pointed to any cogent, supportable reason that would lead a person of ordinary skill in the art to come up with the method of claim 1.

For these reasons, the ‘103 rejection of claim 1 and its dependent claims 2-3, 9 and 23-26 should be withdrawn.

Claim 3 expressly recites that context is saved at a predetermined interruption point. Paragraph 21 of the specification states that a predetermined interruption point can identified if the context switch is performed under software (synchronous) control. Chatterjee clearly states that an interrupt can occur at any time (col. 2, lines 22-23), and that the processor handles interrupts, not a program (col. 5, lines 62-63). Clearly, Chatterjee handles interrupts under hardware control. Thus, the ‘103 rejection of claim 3 should be withdrawn for the additional reason that claim 3 expressly recites a predetermined interruption point, and Chatterjee doesn’t disclose it.

Claim 25 recites that the inconsequential register does not store context at a predetermined interruption point. Claim 26 recites that context is stored in memory other than the inconsequential register. These claims limit the subject matter of claim 1. They were added for clarity, to expressly recite an instance in which context is not stored in an inconsequential register at a predetermined interruption point.

In rejecting both claims, the office action cites passages at col. 3, lines 18-27 and col. 6, line 52 to col. 7, line 30. The rejection is inconsistent, in that the two passages describe different embodiments.

The passage at col. 6, lines 52+ describes how the preferred embodiment functions, specifically how the different privilege levels are assigned to different stack segments, and how stack corruption can be avoided by setting the privilege level of the interrupt handler to a level that is higher than the executing program's privilege level so that a different stack segment is used.

The passage at col. 3, lines 18-27 describes an alternative embodiment, in which return addresses are written to memory locations beyond the stack. In the process, main memory, not register memory, is used as a scratch space for temporary storage. Corruption of the stack is purportedly prevented. The description of the alternative embodiment begins at col. 8, lines 60+.

Nowhere does either passage teach or suggest the features of claims 25 and 26. Moreover, the relevance of these passages to claims 25 and 26 is not clear, and the office action offers no explanation. For these additional reasons, the '103 rejections of claims 25 and 26 should be withdrawn.

Base claim 22 also recites "saving the context under software control using an inconsequential register of the processor." Therefore, the'103 rejection of claim 22 should be withdrawn for the same reasons that the '103 rejection of claim 1 should be withdrawn.

The '103 rejection of base claim 22 should be withdrawn for the additional reason that the Chatterjee does not teach or suggest using an inconsequential register **as temporary storage** for a context switch. Chatterjee only discloses that a stack pointer register can be used for general purposes, such as storing operands and the results of operations (col. 2, lines 65-67).

IV
REJECTION OF CLAIMS 4-5, 11-16 AND 20 UNDER 35 U.S.C §103(b) AS BEING UNPATENTABLE OVER CHATTERJEE U.S. PATENT NO. 5,634,046 IN VIEW OF BUGNION 6,496,847

Argument III is incorporated by reference. Chatterjee does not teach or suggest a context switching method wherein at least one privileged register is used as temporary storage to save the context in other memory at a predetermined interruption point. Chatterjee is silent about handling interrupts under software control in general, and predetermined interrupt points in particular. Chatterjee merely sets the privilege level of the interrupt handler to a level that is higher than the executing program's privilege level.

Bugnion does not teach or suggest using at least one privileged register as temporary storage to save the context in other memory at a predetermined interruption point. Bugnion does not teach or suggest giving access to privileged registers during a context switch and using them as temporary storage.

A passage at col. 11, lines 30-52 gives an overview of Bugnion's context switch. The passage indicates that any available memory may be used to save context. The passage does not indicate whether context is saved at a predetermined interruption point, or whether privileged registers are used as temporary storage to save context.

Therefore, the combined teachings of Chatterjee and Bugnion does not produce the apparatus of claim 11. Chatterjee's Intel processor will use the stack register pointer both for pointing to a stack and for general purposes, and it will prevent the stack from being corrupted when a hardware interrupt occurs.

Bugnion's Virtual Memory Monitor, in the meantime, will run on the processor. The combined teachings do not reach or suggest using at least one privileged register as temporary storage to save the context in the other memory at a predetermined interruption point. Therefore, the '103 rejection of base claim 11 should be withdrawn.

The office action alleges that it would be obvious for Bugnion's Virtual Memory Monitor to handle the interrupts. The problem with that allegation is, the documents made of record don't support it. Chatterjee specifically states that the X86 microprocessor responds to interrupts under hardware control. Bugnion's context switch occurs under asynchronous control (the occurrence of an interrupt – col. 17, lines 6-21). Bugnion is silent about handling interrupts under software control. For this additional reason, the '103 rejection of claim 11 should be withdrawn.

For these same reasons, base claim 12 and its dependent claims 13-16 and 20 should be allowed over the combined teachings of Chatterjee and Bugnion. Base claim 12 also recites "saving the context under software control using an inconsequential register of the processor as temporary storage; and preventing the processor from changing the context while the context is being saved."

V
REJECTION OF CLAIM 22 UNDER 35 U.S.C §101

The office action alleges that claim 22 recites only software and does not recite any associated hardware for execution. The office action also alleges that claim 22 does not require any hardware to implement the claimed invention.

The allegation is not supported by claim 22. Claim 22 explicitly recites a processor and register memory. Therefore, the ‘101 rejection of claim 22 should be withdrawn.

For the reasons above, the rejections should be reversed. The Honorable Board of Patent Appeals and Interferences is respectfully requested to reverse these rejections.

Respectfully submitted,

/Hugh Gortler #33,890/
Hugh P. Gortler, Esq.
Registration No. 33, 890

Hewlett-Packard Company
Intellectual Property Administration
P.O. Box 272400
Fort Collins, Colorado 80527-2400

(949) 454-0898

Date: March 16, 2008

8. CLAIMS APPENDIX

1. (Previously presented) A method of switching context on a processor, the method comprising:

saving the context under software control using an inconsequential register; and

preventing the processor from changing the context while the context is being saved.

2. (Original) The method of claim 1, wherein the inconsequential register is used as a temporary storage in lieu of a privileged register.

3. (Original) The method of claim 1, wherein the context is saved at a predetermined interruption point.

4. (Original) The method of claim 1, wherein the context is switched between a host operating system and a virtual machine application, the virtual machine application controlling the context switch.

5. (Original) The method of claim 4, wherein the inconsequential register is used to pass information to the virtual machine application.

6. (Original) The method of claim 1, wherein the context is switched using an IA-64 processor.

7. (Previously presented) The method of claim 6, wherein the inconsequential register is a caller-save register.

8. (Previously presented) The method of claim 6, wherein the inconsequential register is a branch register.

9. (Original) The method of claim 1, further comprising restoring the context using the inconsequential register as temporary storage.

10. (Original) The method of claim 9, wherein the context is restored by using a branch register to perform an indirect branch.

11. (Previously presented) A method of switching context between a host OS and a virtual machine on a processor, the processor having privileged registers, the processor having access to other memory, the method comprising:

giving the virtual machine access to the privileged registers;

using at least one privileged register as temporary storage to save the context in the other memory at a predetermined interruption point; and

preventing the processor from changing the context while the context is being saved;

the virtual machine application controlling the context switch.

12. (Original) Apparatus comprising:

a processor including a plurality of registers; and

a virtual machine application for commanding the processor to switch context by saving the context under software control using an inconsequential register of the processor as temporary storage; and preventing the processor from changing the context while the context is being saved.

13. (Original) The apparatus of claim 12, wherein the inconsequential register is used as a temporary storage in lieu of a privileged register.

14. (Original) The apparatus of claim 12, wherein the context is saved at a predetermined interruption point.

15. (Original) The apparatus of claim 12, further comprising a host OS; wherein the context is switched between the host OS and the virtual machine application; and wherein the virtual machine application controls the context switch.

16. (Original) The apparatus of claim 15, wherein the inconsequential register is used to pass information to the virtual machine application.

17. (Original) The apparatus of claim 12, wherein the processor is an IA-64 processor.

18. (Previously presented) The apparatus of claim 17, wherein the inconsequential register is a caller-save register.

19. (Previously presented) The apparatus of claim 17, wherein the inconsequential register is a branch register.

20. (Previously presented) The apparatus of claim 12, wherein the virtual application further commands the processor to restore context using the inconsequential register as temporary storage.

21. (Original) The apparatus of claim 20, wherein the context is restored by using a branch register to perform an indirect branch.

22. (Original) Software comprising instructions for commanding a processor to switch context by saving the context under software control using an inconsequential register of the processor as temporary storage; and preventing the processor from changing the context while the context is being saved.

23. (Previously presented) The method of claim 1, wherein content of the inconsequential register is corrupted during the context switch.

24. (Previously presented) The method of claim 1, wherein using the inconsequential register includes storing an address in the inconsequential register, the address indicating a memory location at which the context will be saved.

25. (Previously presented) The method of claim 1, wherein the inconsequential register does not store context at a predetermined interruption point.

26. (Previously presented) The method of claim 1, wherein the context is stored in memory other than the inconsequential register.